Notes on POCKET+ implementation

CONTENTS

MASK UPDATES AND DELTAS

4.2.1 For $t \geq 0$, the build vector is a length-F binary vector defined as

$$\boldsymbol{B_t} = \begin{cases} (\boldsymbol{I_t} \text{ XOR } \boldsymbol{I_{t-1}}) \text{ OR } \boldsymbol{B_{t-1}}, & t > 0,\ \dot{p}_t = 0 \\ \boldsymbol{0}, & \text{otherwise} \end{cases}, \quad (6)$$

where $\dot{p}_t$ is defined in 3.3.2.

4.2.2 For $t > 0$, the mask vector is a length-F binary vector defined as

$$\boldsymbol{M_t} = \begin{cases} (\boldsymbol{I_t} \text{ XOR } \boldsymbol{I_{t-1}}) \text{ OR } \boldsymbol{M_{t-1}}, & \dot{p}_t = 0 \\ (\boldsymbol{I_t} \text{ XOR } \boldsymbol{I_{t-1}}) \text{ OR } \boldsymbol{B_{t-1}}, & \text{otherwise} \end{cases}, \quad (7)$$

where $\dot{p}_t$ is defined in 3.3.2.

NOTE – At $t = 0$, the initial mask vector $\boldsymbol{M_0}$ is user-specified (see 3.3.1).

4.2.3 For $t \geq 0$, the change vector $\boldsymbol{D_t}$ is a length-F binary vector defined as

$$\boldsymbol{D_t} = \begin{cases} \boldsymbol{M_t} \text{ XOR } \boldsymbol{M_{t-1}}, & t > 0 \\ \boldsymbol{0}, & \text{otherwise} \end{cases}. \quad (8)$$

5.3.2 INTERMEDIATE CALCULATIONS

5.3.2.1 For $t \geq 0$, $\dot{d}_t$ is defined as

$$\dot{d}_t = \begin{cases} 1, & \text{if } \dot{f}_t = 0 \text{ and } \dot{r}_t = 0 \\ 0, & \text{otherwise} \end{cases}, \quad (13)$$

where $\dot{f}_t$ and $\dot{r}_t$ are defined in 3.3.2.

5.3.2.2 For $t \geq 0$, V_t is defined as

$$V_t = \begin{cases} R_t, & \text{if } (t - R_t) \leq 0 \\ R_t + C_t, & \text{otherwise} \end{cases}, \quad (14)$$

Notes:
This is the mask update process

Bt is building a new mask so XORing the new and old input vector and adding the result to the existing B vector
If a positive update is required (pt=1) then we reset this to start building again next time

Mt is the present mask it so XORing the new and old input vector and adding the result to the existing mask vector
If a positive update is required (pt=1) then it will XOR the new and old input vector and adding the result to the build vector

You could have a mask saved from before or simply set it to all zeros.

Dt describes the changes in the mask between the now and the last one.

Defines if the output vector will contain no absolute mask and no complete input vector. This is the nominal case, so included as a single bit flag

This increases the robustness period from Rt to Vt. E.g. if there were no mask changes (Dt=0) from the end of the required robustness period to maximum t-16 then Vt>Rt. It's like getting robustness for free. Note you need a tracker to record if Dt was 0 covering the last 16 iterations in order to calculate Ct for this equation.

5.2.2 COUNTER ENCODING FUNCTION

Given a positive integer $1 \le A \le 2^{16} - 1$, the counter encoding function, denoted COUNT(A), maps A onto a variable-length binary vector following table 5-1.

Table 5-1: Counter Encoding Table

Input Integer, A	*Output vector*
$A = 1$	'0'
$2 <= A <= 33$	'110' ‖ $\text{BIT}_5(A-2)$
$A >= 34$	'111' ‖ $\text{BIT}_E(A-2)$

E is calculated as

$$E = 2\lfloor \log_2(A-2) + 1 \rfloor - 6. \quad (9)$$

5.2.3 RUN-LENGTH ENCODING

Given a binary vector, $\boldsymbol{a}$, the run length encoding of $\boldsymbol{a}$, denoted RLE($\boldsymbol{a}$), is a variable-length binary vector defined as

$$\mathbf{RLE}(\boldsymbol{a}) = \text{COUNT}(C_0) || \ldots || \text{COUNT}\left(C_{\text{H}(\boldsymbol{a})-1}\right) || \text{'10'}, \quad (10)$$

where C_i is one more than the number of consecutive '0' bits preceding the ith '1' bit in $\boldsymbol{a}$, starting at the MSB and decreasing.

NOTES

1 If the vector $\boldsymbol{a}$ ends with one or more zeros, they are not explicitly encoded via the COUNT function, as they can be inferred from the length of the input vector and the number of '1' bits, H($\boldsymbol{a}$).

2 When vector $\boldsymbol{a}$ does not contain any '1' bits, the COUNT function will yield a '10' bit string.

0001 000001 1 000001 00001 0000001 ... 1001 0000000000000000

$C_0 = 4$ $C_1 = 6$ $C_2 = 1$ $C_3 = 6$ $C_4 = 5$ $C_5 = 7$ $C_{H(a)-1} = 3$

COUNTERS AND RUN LENGTH ENCODING

Notes:
This is simply a counting system. It is very easy to code and decode. Since zero cannot exist and we use it to code 1, then all the other count values can be reduced by 2 before coding.

L = length of counter field in bits
1 = 0
2 = 11000000 i.e. 110 + 00000 // Special case of zero at position 3, L = 5
33 = 11011111 i.e. 110 + 11111 // Special case of zero at position 3, L = 5
34 = 111100000 i.e. 111 + 100000 // no zero directly follows the 3 zeros, L = 6
65 = 111111111 i.e. 111 + 111111 // no zero directly follows the 3 zeros, L = 6
66 = 11101000000 i.e. 111 + 01000000 // one zero directly follows the 3 zeros, L = 7
And so on, // two zeros directly follow the 3 zeros, L = 8

Simple RLE counting runs of zeros. Terminated by a "10" code word when no more ones are detected. "10" does not exist in the scheme above.

It is easier to find the position of the LOWEST set bit in a word by ANDing it with its own negative value and then using a Debruijn sequence e.g.

```
uint32_t MultiplyDeBruijnBitPosition[32] = {1, 2, 29, 3, 30, 15, 25, 4, 31, 23, 21,
16, 26, 18, 5, 9, 32, 28, 14, 24, 22, 20, 17, 8, 27, 13, 19, 7, 12, 6, 11, 10
};
int x = word $ -word;
int bit_position_in_word = MultiplyDeBruijnBitPosition[((uint32_t)((x * 0x077CB531U))
>> 27]
```

Therefore an actual RLE implementation might count from LSB to MSB. This is the same as applying this function on a bitwise inverted vector.

RECOVERING BIT VALUES FROM SPECIFIC POSITIONS

5.2.4 BIT EXTRACTION FUNCTION

Given two binary vectors $\boldsymbol{a}$, $\boldsymbol{b}$, having the same length, the bit extraction of $\boldsymbol{a}$ relative to $\boldsymbol{b}$, denoted BE($\boldsymbol{a}$, $\boldsymbol{b}$), is the sequence of bits in $\boldsymbol{a}$ taken from the positions where $\boldsymbol{b}$ has a '1' bit. That is,

$$\mathrm{BE}(\boldsymbol{a},\boldsymbol{b}) = \dot{a}_{g_{\mathrm{H}(\boldsymbol{b})-1}} \| \cdots \| \dot{a}_{g_0}, \tag{11}$$

where g_i denotes the position of the ith '1' bit in $\boldsymbol{b}$, starting from the MSB.

Notes:
Runs through vector b starting from the least significant bit, and when it finds a one, it appends the abs value of a bit in the same position in the vector a, into the output stream.

E.g.
a = 0000011000011100
b = 1100001000010100
Output = 10100

5.3.3 OUTPUT VECTOR COMPONENTS

5.3.3.1 The binary vector $\boldsymbol{h_t}$ is defined as

$$\boldsymbol{h_t} = \text{RLE}(\boldsymbol{X_t}) \parallel \text{BIT}_4(V_t) \parallel \boldsymbol{e_t} \parallel \boldsymbol{k_t} \parallel \boldsymbol{c_t} \parallel \dot{d}_t, \tag{15}$$

where V_t is defined in 5.3.2.2, $\dot{d}_t$ is defined in 5.3.2.1, and $\boldsymbol{X_t}$, $\boldsymbol{e_t}$, $\boldsymbol{c_t}$, and $\boldsymbol{k_t}$ are defined as

$$\boldsymbol{X_t} = \begin{cases} < \boldsymbol{D_t} >, & \text{if } R_t = 0 \\ < (\boldsymbol{D_1} \text{ OR } \boldsymbol{D_2} \text{ OR} \ldots \text{OR } \boldsymbol{D_t}) >, & \text{if } (t - R_t) \leq 0 \\ < (\boldsymbol{D_{t-R_t}} \text{ OR } \boldsymbol{D_{t-R_t+1}} \text{ OR} \ldots \text{OR } \boldsymbol{D_t}) >, & \text{otherwise} \end{cases} \tag{16}$$

where $\boldsymbol{D_t}$ is the change binary vector defined in 4.2.3, and R_t is defined in 3.3.2,

$$\boldsymbol{y_t} = \text{BE}(< \sim \boldsymbol{M_t} >, \boldsymbol{X_t}), \tag{17}$$

$$\boldsymbol{e_t} = \begin{cases} \emptyset, & \text{if } V_t = 0 \text{ or } \boldsymbol{X_t} = \boldsymbol{0} \\ \text{'0'}, & \text{if } \boldsymbol{y_t} = \boldsymbol{0} \text{ and } V_t > 0 \text{ and } \boldsymbol{X_t} \neq \boldsymbol{0} \\ \text{'1'}, & \text{otherwise} \end{cases}, \tag{18}$$

$$\boldsymbol{k_t} = \begin{cases} \emptyset, & \text{if } V_t = 0 \text{ or } \boldsymbol{X_t} = \boldsymbol{0} \text{ or } \boldsymbol{y_t} = \boldsymbol{0} \\ \boldsymbol{y_t}, & \text{otherwise} \end{cases}, \tag{19}$$

$$\boldsymbol{c_t} = \begin{cases} \emptyset, & \text{if } \boldsymbol{k_t} = \emptyset \\ \text{'0'}, & \text{if } \boldsymbol{k_t} \neq \emptyset \text{ and } \dot{p}_i \text{ is not set to one more than once, for} \\ & i \text{ in } \{\max(0, t - V_t), \max(0, t - V_t) + 1, \ldots, t - 1, t\}, \\ \text{'1'}, & \text{otherwise} \end{cases}. \tag{20}$$

5.3.3.2 The binary vector $\boldsymbol{q_t}$ is defined as

$$\boldsymbol{q_t} = \begin{cases} \emptyset, & \text{if } \dot{d}_t = 1 \\ \text{'1'} \parallel \text{RLE}(< (\boldsymbol{M_t} \text{ XOR } (\boldsymbol{M_t}^{\ll})) >), & \text{if } \dot{f}_t = 1 \\ \text{'0'}, & \text{otherwise} \end{cases}, \tag{21}$$

where $\dot{f}_t$ is defined in 3.3.2, and $\dot{d}_t$ is defined in 5.3.2.1.

CODING OF THE OUTPUT VECTOR

Notes:
Just shows the overall structure of ht. Vt is the enhanced robustness level. The usual case is RLE(Xt), Vt, et and dt.

Xt is all the mask changes in the robustness time span used, only complicated mathematically to cover the simple cases of robustness is zero and the initialisation phase. To see why it is inverted see the notes on RLE encoding.

yt shows the order of all positive and negative mask changes in Xt as 1s and 0s. Note it is written "forwards" i.e. from most significant to least significant bit, as although Mt is inverted here, and Xt is already inverted, the BE function reads from least to most significant bit.

et=1, indicates yt will follow, the other cases are

- no need as Vt=0 so it's simply the inverse of what was before or Xt was zero so no changes at all
- all mask changes were negative

kt is simply yt if case 3 of et was activated

ct decides if a bit had more than one positive mask change in the robustness period. This is bad since you cannot work out the abs value of the associated bit and therefore all positive change abs bit values will have to be sent as part of unpredictable bits values sent later.

dt, states if any entire masks or entire input vectors need to be sent. If one (i.e. no) then the output vector is finished.

qt is present mask XORed with itself bit shifted to the left by one. This allows for efficient RLE since we usually see rows of ones and zeros in the mask. The first bit simply indicates if a mask follows or not if dt was set to zero (as it could be mask only or abs bits only). To see why the masks are inverted see the notes on RLE encoding.

SPECIAL NOTE ON EQ 20

$$c_t = \begin{cases} \emptyset, & \text{if } \boldsymbol{k}_t = \emptyset \\ \text{'0'}, & \text{if } \boldsymbol{k}_t \neq \emptyset \text{ and } \dot{p}_i \text{ is not set to one more than once, for} \\ & i \text{ in } \{\max(0, t - V_t), \max(0, t - V_t) + 1, \ldots, t - 1, t\}, \\ \text{'1'}, & \text{otherwise} \end{cases} \quad (20)$$

Notes:
ct decides if more than one positive mask change occurs in the robustness period, Vt. This is bad since you cannot work out the actual value of the associated bit, see diagram below. Since this is a rare case, it was decided to send all the actual values of all the bits that are flagged as positive changes in the mask. See, Eq 22 and the condition relying on ct in that equation.

Predictable Value = 1 — 111110011 | 111111111111 | 11111111 — Predictable Value = 1

+ Update + Update

Predictable Value = 1 — 111110000 | 000000000000 | 000000000 — Predictable Value = 0

+ Update + Update

5.3.3.3 The binary vector $\boldsymbol{u}_t$, is defined as

$$\boldsymbol{u}_t = \begin{cases} \text{BE}\left(\boldsymbol{I}_t, (< \boldsymbol{X}_t > \ OR\ \boldsymbol{M}_t)\right), & \text{if } \dot{d}_t = 1 \text{ and } \boldsymbol{c}_t = 1 \\ \text{BE}(\boldsymbol{I}_t, \boldsymbol{M}_t), & \text{if } \dot{d}_t = 1 \text{ and } \boldsymbol{c}_t \neq 1 \\ '1' \parallel \text{COUNT}(F) \parallel \boldsymbol{I}_t, & \text{if } \dot{r}_t = 1 \\ '0' \parallel \text{BE}\left(\boldsymbol{I}_t, (< \boldsymbol{X}_t > \ OR\ \boldsymbol{M}_t)\right), & \text{if } \dot{r}_t = 0 \text{ and } \dot{f}_t = 1 \text{ and } \boldsymbol{c}_t = 1 \\ '0' \parallel \text{BE}(\boldsymbol{I}_t, \boldsymbol{M}_t), & \text{otherwise,} \end{cases} \quad (22)$$

where $\dot{r}_t$ and $\dot{f}_t$ are defined in 3.3.2, $\boldsymbol{c}_t$ is defined in 5.3.3.1, and $\dot{d}_t$ is defined in 5.3.2.1.

5.3.3.3 The binary vector $\boldsymbol{u_t}$, is defined as

$$\boldsymbol{u_t} = \begin{cases} \mathrm{BE}\left(\boldsymbol{I_t}, (< \boldsymbol{X_t} > \ OR\ \boldsymbol{M_t})\right), & \text{if } \dot{d}_t = 1 \text{ and } \boldsymbol{c_t} = 1 \\ \mathrm{BE}(\boldsymbol{I_t}, \boldsymbol{M_t}), & \text{if } \dot{d}_t = 1 \text{ and } \boldsymbol{c_t} \neq 1 \\ '1' \parallel \mathrm{COUNT}(F) \parallel \boldsymbol{I_t}, & \text{if } \dot{r}_t = 1 \\ '0' \parallel \mathrm{BE}\left(\boldsymbol{I_t}, (< \boldsymbol{X_t} > \ OR\ \boldsymbol{M_t})\right), & \text{if } \dot{r}_t = 0 \text{ and } \dot{f}_t = 1 \text{ and } \boldsymbol{c_t} = 1 \\ '0' \parallel \mathrm{BE}(\boldsymbol{I_t}, \boldsymbol{M_t}), & \text{otherwise,} \end{cases} , \quad (22)$$

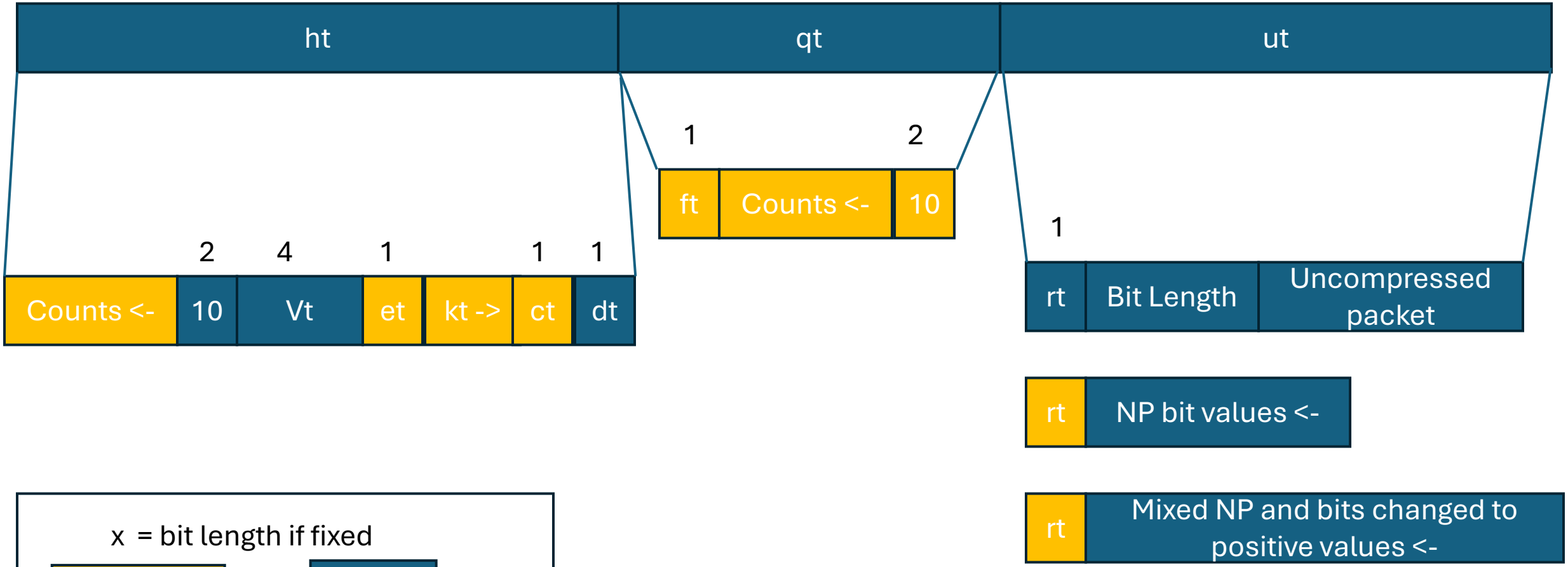
where $\dot{r}_t$ and $\dot{f}_t$ are defined in 3.3.2, $\boldsymbol{c_t}$ is defined in 5.3.3.1, and $\dot{d}_t$ is defined in 5.3.2.1.

Notes:
ut sends absolute values of bits of the input vector. Described in order of complication of cases

- Case 2: Sends only the abs bit values of the unpredictable bits
- Case 1: Sends the abs bit values of the unpredictable bits and all the abs bit values of the bits that changed positively in the mask (as ct=1)
- Case 5: Sends a zero and then same as case 2
- Case 4: Sends a zero and then same as case 1
- Case 3: Sends a one, followed by the length of the input vector in bits encoded using table 5-1, then the entire input vector in the clear.

Here Xt has to be inverted into the same direction as Mt, as the BE function will read them both from the least to the most significant bit i.e. ut is “backwards”

DIAGRAM OF THE OUTPUT VECTOR



x = bit length if fixed

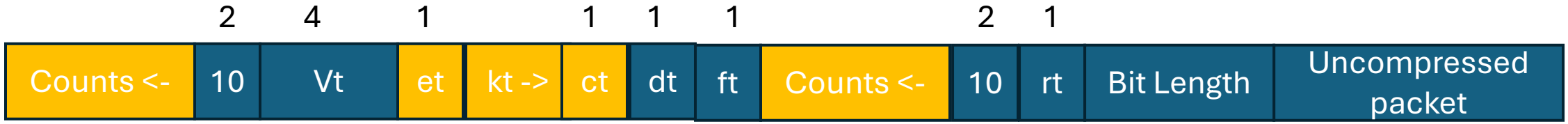
Might not exist

Exists

<- = written from LSBit to MSbit

-> = written from MSBit to LSbit

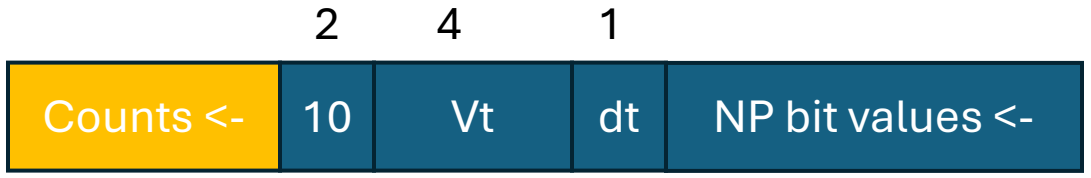
INITIALISATION PACKET, sent for the first Rt+1 iterations



Where dt = “0”, ft = “1”, rt = “1”,

If a reliable transfer protocol is available (e.g. FTP or TCP/IP), you could force VT to 0 to simplify on-board and ground processing

After initialisation the compressed packets are always as shown below
No need to send full masks or uncompressed packets periodically



Where Vt =“0000” and dt “1”

Different sync recovery mechanisms

Consider a GAP as from the end of the Vt range in the present packet to the occurrence of the last packet successfully decompressed.

Gap	Pt=1 occurrences in the gap?	Sync Recovery required	Why
<1	N/A	None	Ground mask update covers the missing period
>0	1	Resend the mask	The ground mask update is not covering the missing period but the reference packet predictable values are still valid
>0	>1	Resend the mask (might be enough), if not, request a packet in the clear	The ground mask update is not covering the missing period and the reference packet predictable values might* be invalid now

- Consider this; in the last received packet a bit had the value one and was NP. Then pt=1 and it became stable at a value zero. Then pt = 1 and it became predictable. So it is not in the NP bits put the transmitter assumes a value one and the receiver zero. NOTE: if this does not happen to any bit then just a mask resend alone will resync you.

Note that because masks are compressed before sending (and usually achieve a compression ratio of 3-4) the impact of resending masks is not significant. As this will cover the majority of the resync cases above, it is recommended that this is done more often than resending a packet in the clear periodically.

Also note that if synchronisation is lost, compressed packets can be saved and then decompressed once a resynchronisation is achieved in the future. This is because the mask changes can be propagated backwards and well as forwards in time.

DEALING WITH MISSING PACKETS

The standard does not include a method for detecting missing packets. So how does the decoder know if the gap > Vt?
There are several options

1) Create a packet counter and attach it to each compressed packet as meta data
2) Wrap the entire compressed packet in another CCSDS packet which contains a packet sequence count, see below
3) Try to decompress the packet and see if the source sequence count of the uncompressed packet updates as expected
4) Try to decompress the packet and check the result against an uncompressed packet CRC

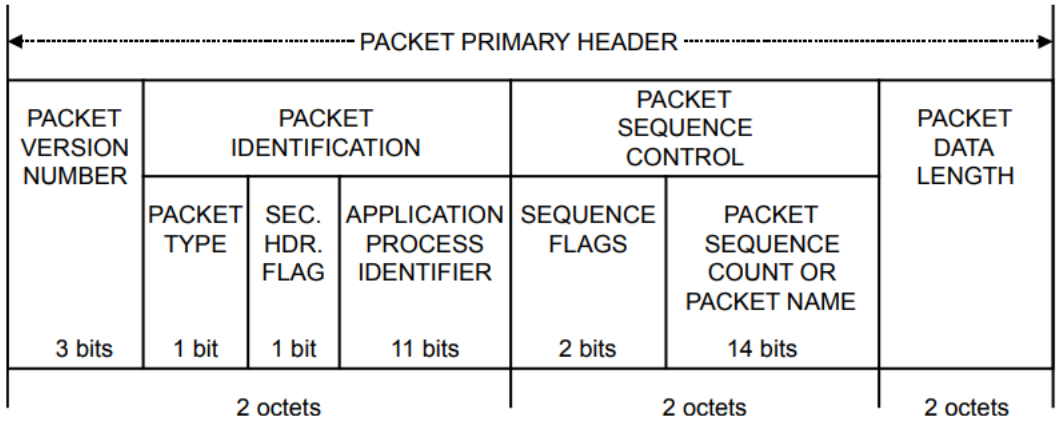


Figure 4-2: Packet Primary Header

Dealing with corrupt or unsynchronised packets

The standard does not tell you how to deal with corrupt packets. A single check that catches the majority of problems is that the number of NP bit values sent by the compressor corresponds to the number of NP positions in the ground mask. One could add that number as meta data for each packet, or try to use some of the bit stuffing field by adding a three-bit counter (0-7) to the end of the compressed packet which identifies the number of bits that would have been stuffed to reach the byte boundary. One could also add a NP field parity bit to the above scheme.

Note in the paper "**Implementing the New CCSDS Housekeeping Data Compression Standard 124.0-B-1 (based on POCKET+) on OPS-SAT-1" August 2021.** there are other examples of checks that can be done to ensure integrity of the compressed packet. However, none of the them provides 100% confidence on their own, even implementing a packet CRC to check the decompressed packet. This is because we can have the situation where the mask can be wrong but the impacted NP bits are in the same state as in the last uncompressed packet. We have a good reference packet (as it is correct) but the ground mask is wrong going forward.

On the other hand, if corruption occurs in the NP bit value field then it will not impact the decompression going forward, and implementing several checks on integrity provides significant overall confidence. Finally, if a reliable transfer protocol is used then this also eliminates the need for these checks.

DEALING WITH MULTIPLE BIT STREAMS BEING COMPRESSED IN PARALLEL

The standard does ot tell you how to deal with multiple bit streams being compressed in parallel. There are several ways to deal with this

1) Create a packet ID and attach it to each compressed packet as meta data
2) Create separate files that only contain the compressed packets from one bit stream and can be identified by the file name
3) Preprocess a file containing compressed packets from multiple bit streams into the compressed packets from one bit stream sequentially following the next. The initialisation process at the start of each bit stream processing will ensure that the whole file can be decompressed separately on the ground without any special processing.

PUESDO CODE FOR DECOMPRESSION

READ IN THE COMPRESSED FILE AS A BYTE ARRAY

FIND THE BIT LENGTH OF THE UNCOMPRESSED PACKET TO SET THE BUFFER SIZES

1. Run through the input byre array to arrive at the bit length field in ut, read and convert it to packetSize in bytes
2. Create the necessary packet buffers, I, M, X_Pos using packetSize. I and M are state vectors and not yet set
3. Return to the start of the input byte array

START OF PACKET PROCESSING LOOP

PROCESSING ht

1. Set ct equal to zero
2. Read first two bits and if there are no delta mask counters in ht, move directly to reading Vt
3. If there are delta mask counters in ht, then process them and update the mask buffer, M, accordingly, if ct=1 then record all the positive mask change positions in the X_pos buffer (as needed in the ut processing)
4. Read Vt and dt

PROCESSING qt

4. If dt = 1, then set ft and rt to zero, there is no entire mask included (qt) so go to ut processing
5. If dt = 0, then check the value of ft
6. If ft = 1, then process the counters and replace the mask buffer M with the new one

PROCESSING ut

7. If dt = 1, rt is already known as zero, otherwise it is read
8. If rt = 1, then read the bit length and convert to packetSize in bytes, the output buffer I is replaced with the uncompressed packet field; then I is output to file and loop is restarted
9. if rt = 0, then check the value of ct
10. If ct = 0 then the non predictable bit field is read, then using the mask, the corresponding masked bit values are set in the output buffer I; then I is output to file and loop is restarted
11. if ct = 1 then the mask and the X_pos buffer are ORed and the corresponding bit values are set in the output buffer I as the non predictable bit field is read; then I is output to file and loop is restarted

END OF PACKET PROCESSING LOOP